

Architecture de Plugins pour BlueNotebook - Spécification Complète

1. Objectifs et Principes Fondamentaux

1.1 Principes Clés

1. **Découplage Fort** : Aucune dépendance directe entre l'application et les plugins spécifiques
 2. **Découverte Automatique** : Chargement dynamique sans modification du code principal
 3. **API Claire et Stable** : Contrat bien défini avec versioning sémantique
 4. **Gestion du Cycle de Vie** : Contrôle complet du chargement/déchargement
 5. **Extensibilité des Préférences** : Configuration intégrée pour chaque plugin
 6. **Isolation des Erreurs** : Un plugin défaillant ne doit pas crasher l'application
 7. **Performance** : Chargement paresseux (lazy loading) des plugins non utilisés
-

2. Structure d'un Plugin

2.1 Organisation des Fichiers

```
bluenotebook/  
├── plugins/  
│   ├── meteo/  
│   │   ├── __init__.py      # Point d'entrée  
│   │   ├── plugin.json     # Métadonnées  
│   │   ├── preferences_widget.py  # Widget de configuration (optionnel)  
│   │   ├── requirements.txt  # Dépendances Python (optionnel)  
│   │   ├── assets/         # Ressources (icônes, traductions...)  
│   │   │   ├── icon.png  
│   │   │   └── translations/
```

```

| | | └── fr.json
| | | └── en.json
| | └── tests/          # Tests unitaires du plugin
| |     └── test_meteo.py
| |
| └── carte_gps/
|     ├── __init__.py
|     ├── plugin.json
|     └── ...
└── core/
    ├── plugin_manager.py    # Gestionnaire principal
    ├── plugin_api.py        # API et classe de base
    ├── plugin_loader.py     # Chargement dynamique
    ├── plugin_registry.py   # Registre des plugins
    └── plugin_exceptions.py  # Exceptions personnalisées

```

2.2 Fichier de Métadonnées ((plugin.json))

json

```

{
  "name": "Météo du Jour",
  "id": "com.bluenotebook.meteo",
  "version": "1.0.0",
  "api_version": "1.0",
  "author": "Jean-Marc DIGNE",
  "email": "contact@example.com",
  "description": "Insère la météo actuelle pour une ville donnée.",
  "description_long": "Ce plugin permet d'insérer automatiquement...",
  "homepage": "https://github.com/user/bluenotebook-meteo",
  "license": "GPL-3.0",
  "entry_point": "meteo:MeteoPlugin",
  "dependencies": {
    "python": ">=3.8",
    "bluenotebook": ">=2.0.0",
    "packages": ["requests>=2.28.0"]
  },
  "permissions": [
    "network",
    "editor.insert",
    "preferences.add"
  ],
  "categories": ["productivity", "weather"],
  "keywords": ["météo", "weather", "température"],
  "icon": "assets/icon.png",
  "enabled_by_default": true,
  "min_qt_version": "5.15"
}

```

Champs obligatoires et optionnels

Obligatoires :

- (name), (id), (version), (author), (entry_point), (api_version)

Optionnels :

- `email`, `description_long`, `homepage`, `license`, `dependencies`, `permissions`, `categories`, `keywords`, `icon`, `enabled_by_default`, `min_qt_version`
-

3. API du Plugin (`plugin_api.py`)

3.1 Classe de Base

```
python
```

```
# bluenotebook/core/plugin_api.py
```

```
from abc import ABC, abstractmethod
```

```
from typing import List, Optional, Tuple
```

```
from PyQt6.QtWidgets import QAction, QWidget
```

```
from PyQt6.QtCore import QObject, pyqtSignal
```

```
class PluginState:
```

```
    """États possibles d'un plugin"""
```

```
    DISCOVERED = "discovered"
```

```
    LOADED = "loaded"
```

```
    INITIALIZED = "initialized"
```

```
    ACTIVE = "active"
```

```
    ERROR = "error"
```

```
    DISABLED = "disabled"
```

```
class BlueNotebookPlugin(QObject, ABC):
```

```
    """Classe de base abstraite pour tous les plugins."""
```

```
# Signaux pour la communication plugin -> application
```

```
status_changed = pyqtSignal(str) # Émis lors d'un changement d'état
```

```
error_occurred = pyqtSignal(str, Exception) # Émis en cas d'erreur
```

```
def __init__(self, plugin_manager):
```

```
    super().__init__()
```

```
    self.plugin_manager = plugin_manager
```

```
    self.main_window = plugin_manager.main_window
```

```
    self.state = PluginState.DISCOVERED
```

```
    self._metadata = {}
```

```
    self._settings = {}
```

```
# Propriétés en lecture seule
```

```
@property
```

```
def metadata(self):
```

```
    """Métadonnées du plugin (depuis plugin.json)"""
```

```
    return self._metadata
```

```
@property
```

```
def name(self):
    return self._metadata.get('name', 'Unknown')

@property
def version(self):
    return self._metadata.get('version', '0.0.0')

@property
def id(self):
    return self._metadata.get('id', 'unknown')

# Méthodes du cycle de vie (optionnelles)
def on_load(self):
    """Appelé après le chargement du module, avant l'initialisation."""
    pass

@abstractmethod
def on_initialize(self):
    """
    Appelé une fois au démarrage. Initialisation des ressources.
    DOIT être implémenté par tous les plugins.
    """
    pass

def on_activate(self):
    """Appelé lorsque le plugin est activé (après initialisation)."""
    pass

def on_deactivate(self):
    """Appelé lorsque le plugin est désactivé."""
    pass

def on_shutdown(self):
    """Appelé à la fermeture de l'application. Nettoyage des ressources."""
    pass

# Enregistrement des composants
```

```
def register_actions(self) -> List[QAction]:
```

```
    """
```

Retourne une liste d'actions à ajouter au menu 'Intégrations'.

Ces actions seront automatiquement ajoutées/retirées lors de l'activation/désactivation.

```
    """
```

```
    return []
```

```
def register_toolbar_items(self) -> List[QAction]:
```

```
    """Retourne une liste d'actions pour la barre d'outils."""
```

```
    return []
```

```
def register_preferences_widget(self) -> Optional[Tuple[str, QWidget]]:
```

```
    """
```

Retourne un tuple (nom_onglet, QWidget) pour les Préférences.

Le widget doit implémenter les méthodes save_settings() et load_settings().

```
    """
```

```
    return None
```

```
def register_shortcuts(self) -> dict:
```

```
    """
```

Retourne un dictionnaire de raccourcis clavier.

Format: {'action_name': 'Ctrl+Shift+M'}

```
    """
```

```
    return {}
```

API d'accès aux fonctionnalités de BlueNotebook

```
def get_editor(self):
```

```
    """Retourne l'interface sécurisée de l'éditeur."""
```

```
    return self.plugin_manager.get_editor_api()
```

```
def get_journal(self):
```

```
    """Retourne l'interface d'accès au journal."""
```

```
    return self.plugin_manager.get_journal_api()
```

```
def get_settings(self, key: str, default=None):
```

```
    """Récupère une valeur de configuration du plugin."""
```

```
    return self._settings.get(key, default)
```

```
def set_settings(self, key: str, value):
```

```
    """Sauvegarde une valeur de configuration du plugin."""
```

```
    self._settings[key] = value
```

```
    self.plugin_manager.save_plugin_settings(self.id, self._settings)
```

```
def log_info(self, message: str):
```

```
    """Enregistre un message d'information dans les logs."""
```

```
    self.plugin_manager.log(f"[{self.name}] {message}", level="INFO")
```

```
def log_error(self, message: str, exception: Exception = None):
```

```
    """Enregistre une erreur dans les logs."""
```

```
    self.plugin_manager.log(f"[{self.name}] {message}", level="ERROR", exception=exception)
```

```
def show_notification(self, title: str, message: str, duration: int = 3000):
```

```
    """Affiche une notification à l'utilisateur."""
```

```
    self.plugin_manager.show_notification(title, message, duration)
```

```
class EditorAPI:
```

```
    """Interface sécurisée pour interagir avec l'éditeur."""
```

```
def __init__(self, editor_widget):
```

```
    self._editor = editor_widget
```

```
def get_text(self) -> str:
```

```
    """Retourne tout le texte de l'éditeur."""
```

```
    return self._editor.toPlainText()
```

```
def get_selected_text(self) -> str:
```

```
    """Retourne le texte sélectionné."""
```

```
    cursor = self._editor.textCursor()
```

```
    return cursor.selectedText()
```

```
def insert_text_at_cursor(self, text: str):
```

```
    """Insère du texte à la position du curseur."""
```

```
    cursor = self._editor.textCursor()
```



```
cursor.insertText(text)
```

```
def replace_selection(self, text: str):  
    """Remplace la sélection par le texte fourni."""  
    cursor = self._editor.textCursor()  
    cursor.insertText(text)  
  
def get_cursor_position(self) -> Tuple[int, int]:  
    """Retourne (ligne, colonne) de la position du curseur."""  
    cursor = self._editor.textCursor()  
    return (cursor.blockNumber(), cursor.columnNumber())
```

```
def set_cursor_position(self, line: int, column: int):  
    """Déplace le curseur à la position spécifiée."""  
    # Implementation...  
    pass
```

```
class JournalAPI:  
    """Interface sécurisée pour interagir avec le journal."""  
  
    def __init__(self, journal):  
        self._journal = journal  
  
    def get_current_date(self):  
        """Retourne la date courante affichée."""  
        return self._journal.current_date  
  
    def get_entry_for_date(self, date):  
        """Retourne l'entrée pour une date donnée."""  
        return self._journal.get_entry(date)  
  
    def search_entries(self, query: str) -> list:  
        """Recherche dans les entrées du journal."""  
        return self._journal.search(query)
```

4. Gestionnaire de Plugins (`plugin_manager.py`)

4.1 Implémentation Complète

python

```
# bluenotebook/core/plugin_manager.py
```

```
import os
```

```
import sys
```

```
import json
```

```
import logging
```

```
import importlib.util
```

```
from typing import Dict, List, Optional
```

```
from pathlib import Path
```

```
from .plugin_api import BlueNotebookPlugin, PluginState, EditorAPI, JournalAPI
```

```
from .plugin_exceptions import (
```

```
    PluginLoadError, PluginInitError, PluginDependencyError,
```

```
    PluginVersionError, PluginPermissionError
```

```
)
```

```
class PluginManager:
```

```
    """Gestionnaire central des plugins."""
```

```
    API_VERSION = "1.0"
```

```
    def __init__(self, main_window, config_manager):
```

```
        self.main_window = main_window
```

```
        self.config_manager = config_manager
```

```
        self.plugins: Dict[str, BlueNotebookPlugin] = {}
```

```
        self.plugins_metadata: Dict[str, dict] = {}
```

```
        self.plugins_dir = self._get_plugins_directory()
```

```
        self.logger = logging.getLogger('BlueNotebook.PluginManager')
```

```
        # APIs exposées aux plugins
```

```
        self.editor_api = None
```

```
        self.journal_api = None
```

```
    def _get_plugins_directory(self) -> Path:
```

```
        """Détermine le chemin du répertoire des plugins."""
```

```
        # Ordre de priorité :
```

```
        # 1. Répertoire utilisateur ~/.bluenotebook/plugins/
```

```
        # 2. Répertoire de l'application
```

```
        user_plugins = Path.home() / '.bluenotebook' / 'plugins'
```

```
app_plugins = Path(__file__).parent.parent / 'plugins'
```

```
if user_plugins.exists():
```

```
    return user_plugins
```

```
return app_plugins
```

```
def initialize(self, editor_widget, journal):
```

```
    """Initialise le gestionnaire avec les composants de l'application."""
```

```
    self.editor_api = EditorAPI(editor_widget)
```

```
    self.journal_api = JournalAPI(journal)
```

```
def discover_plugins(self) -> List[str]:
```

```
    """
```

```
    Scanne le répertoire des plugins et retourne la liste des IDs découverts.
```

```
    Ne charge pas encore les plugins.
```

```
    """
```

```
    discovered = []
```

```
if not self.plugins_dir.exists():
```

```
    self.logger.warning(f"Répertoire plugins non trouvé : {self.plugins_dir}")
```

```
    return discovered
```

```
for item in self.plugins_dir.iterdir():
```

```
    if not item.is_dir() or item.name.startswith('_):
```

```
        continue
```

```
    metadata_file = item / 'plugin.json'
```

```
if not metadata_file.exists():
```

```
    self.logger.warning(f"Plugin '{item.name}' ignoré : pas de plugin.json")
```

```
    continue
```

```
try:
```

```
    with open(metadata_file, 'r', encoding='utf-8') as f:
```

```
        metadata = json.load(f)
```

```
# Validation des champs obligatoires
```

```
required = ['name', 'id', 'version', 'entry_point', 'api_version']
```

```
missing = [field for field in required if field not in metadata]
```

```
if missing:
```

```
    raise ValueError(f'Champs manquants : {missing}')
```

```
# Vérification de la compatibilité API
```

```
if metadata['api_version'] != self.API_VERSION:
```

```
    raise PluginVersionError(
```

```
        f'API incompatible : plugin={metadata['api_version']}, "
```

```
        f'app={self.API_VERSION}"
```

```
    )
```

```
plugin_id = metadata['id']
```

```
self.plugins_metadata[plugin_id] = metadata
```

```
self.plugins_metadata[plugin_id]['_path'] = item
```

```
discovered.append(plugin_id)
```

```
self.logger.info(f'Plugin découvert : {metadata['name']} v {metadata['version']}')
```

```
except Exception as e:
```

```
    self.logger.error(f'Erreur lors de la découverte de '{item.name}': {e}')
```

```
return discovered
```

```
def load_plugin(self, plugin_id: str) -> bool:
```

```
    """
```

```
    Charge dynamiquement un plugin spécifique.
```

```
    Retourne True si succès, False sinon.
```

```
    """
```

```
if plugin_id in self.plugins:
```

```
    self.logger.warning(f'Plugin '{plugin_id}' déjà chargé')
```

```
    return True
```

```
if plugin_id not in self.plugins_metadata:
```

```
    raise PluginLoadError(f'Plugin '{plugin_id}' non découvert')
```

```
metadata = self.plugins_metadata[plugin_id]
```

```
plugin_path = metadata['_path']
```

try:

Vérifier les dépendances

self._check_dependencies(metadata)

Charger le module

entry_point = metadata['entry_point']

module_name, class_name = entry_point.split(':')

spec = importlib.util.spec_from_file_location(

 module_name,

 plugin_path / '__init__.py'

)

module = importlib.util.module_from_spec(spec)

sys.modules[module_name] = module

spec.loader.exec_module(module)

Instancier la classe du plugin

plugin_class = getattr(module, class_name)

if not issubclass(plugin_class, BlueNotebookPlugin):

 raise PluginLoadError(

 f"La classe {class_name} doit hériter de BlueNotebookPlugin"

)

plugin_instance = plugin_class(self)

plugin_instance._metadata = metadata

plugin_instance.state = PluginState.LOADED

Charger les paramètres sauvegardés

saved_settings = self.config_manager.get_plugin_settings(plugin_id)

plugin_instance._settings = saved_settings or {}

Appeler le hook on_load

plugin_instance.on_load()

self.plugins[plugin_id] = plugin_instance

self.logger.info(f"Plugin '{metadata['name']}' chargé avec succès")

```
return True
```

```
except Exception as e:
```

```
    self.logger.error(f"Erreur lors du chargement de '{plugin_id}': {e}")
```

```
    raise PluginLoadError(f"Impossible de charger '{plugin_id}': {e}")
```

```
def initialize_plugin(self, plugin_id: str) -> bool:
```

```
    """Initialise un plugin chargé."""
```

```
    if plugin_id not in self.plugins:
```

```
        raise PluginInitError(f"Plugin '{plugin_id}' non chargé")
```

```
    plugin = self.plugins[plugin_id]
```

```
    try:
```

```
        plugin.on_initialize()
```

```
        plugin.state = PluginState.INITIALIZED
```

```
        self.logger.info(f"Plugin '{plugin.name}' initialisé")
```

```
    return True
```

```
except Exception as e:
```

```
    plugin.state = PluginState.ERROR
```

```
    self.logger.error(f"Erreur lors de l'initialisation de '{plugin_id}': {e}")
```

```
    plugin.error_occurred.emit(str(e), e)
```

```
    return False
```

```
def activate_plugin(self, plugin_id: str) -> bool:
```

```
    """Active un plugin initialisé."""
```

```
    if plugin_id not in self.plugins:
```

```
        return False
```

```
    plugin = self.plugins[plugin_id]
```

```
    if plugin.state != PluginState.INITIALIZED:
```

```
        self.logger.warning(
```

```
            f"Plugin '{plugin_id}' doit être initialisé avant activation"
```

```
        )
```

```
    return False
```

try:

Enregistrer les actions dans le menu

actions = plugin.register_actions()

for action in actions:

self.main_window.integrations_menu.addAction(action)

Enregistrer le widget de préférences

pref_info = plugin.register_preferences_widget()

if pref_info:

tab_name, widget = pref_info

self.main_window.preferences_dialog.add_plugin_tab(
 plugin_id, tab_name, widget
)

Appeler le hook d'activation

plugin.on_activate()

plugin.state = PluginState.ACTIVE

self.logger.info(f"Plugin '{plugin.name}' activé")

return True

except Exception as e:

plugin.state = PluginState.ERROR

self.logger.error(f"Erreur lors de l'activation de '{plugin_id}' : {e}")

return False

def deactivate_plugin(self, plugin_id: str):

"""Désactive un plugin actif."""

if plugin_id not in self.plugins:

return

plugin = self.plugins[plugin_id]

try:

plugin.on_deactivate()

plugin.state = PluginState.INITIALIZED

Retirer les actions du menu, etc.


```
except Exception as e:
```

```
    self.logger.error(f"Erreur lors de la désactivation de '{plugin_id}' : {e}")
```

```
def load_enabled_plugins(self):
```

```
    """Charge et active tous les plugins marqués comme activés."""
```

```
    enabled_plugins = self.config_manager.get_enabled_plugins()
```

```
    for plugin_id in self.plugins_metadata.keys():
```

```
        # Vérifier si activé par défaut ou par l'utilisateur
```

```
        enabled_by_default = self.plugins_metadata[plugin_id].get('enabled_by_default', False)
```

```
        is_enabled = plugin_id in enabled_plugins or enabled_by_default
```

```
        if is_enabled:
```

```
            try:
```

```
                self.load_plugin(plugin_id)
```

```
                self.initialize_plugin(plugin_id)
```

```
                self.activate_plugin(plugin_id)
```

```
            except Exception as e:
```

```
                self.logger.error(f"Impossible d'activer '{plugin_id}' : {e}")
```

```
def shutdown_all_plugins(self):
```

```
    """Arrête proprement tous les plugins."""
```

```
    for plugin_id, plugin in self.plugins.items():
```

```
        try:
```

```
            plugin.on_shutdown()
```

```
            self.logger.info(f"Plugin '{plugin.name}' arrêté")
```

```
        except Exception as e:
```

```
            self.logger.error(f"Erreur lors de l'arrêt de '{plugin_id}' : {e}")
```

```
def _check_dependencies(self, metadata: dict):
```

```
    """Vérifie que toutes les dépendances sont satisfaites."""
```

```
    deps = metadata.get('dependencies', {})
```

```
    # Vérifier la version de Python
```

```
    if 'python' in deps:
```

```
        # Implementation de la vérification de version...
```

```
    pass
```

```

# Vérifier les packages Python
if 'packages' in deps:
    for package in deps['packages']:
        # Vérifier si le package est installé
        pass

# Méthodes exposées aux plugins
def get_editor_api(self) -> EditorAPI:
    return self.editor_api

def get_journal_api(self) -> JournalAPI:
    return self.journal_api

def save_plugin_settings(self, plugin_id: str, settings: dict):
    """Sauvegarde les paramètres d'un plugin."""
    self.config_manager.save_plugin_settings(plugin_id, settings)

def log(self, message: str, level: str = "INFO", exception: Exception = None):
    """Enregistre un message dans les logs."""
    log_func = getattr(self.logger, level.lower())
    if exception:
        log_func(message, exc_info=exception)
    else:
        log_func(message)

def show_notification(self, title: str, message: str, duration: int = 3000):
    """Affiche une notification système."""
    # Utiliser QSystemTrayIcon ou une notification Qt
    pass

```

5. Exceptions Personnalisées ((plugin_exceptions.py))

python

```
# bluenotebook/core/plugin_exceptions.py
```

```
class PluginError(Exception):
    """Classe de base pour toutes les erreurs de plugin."""
    pass

class PluginLoadError(PluginError):
    """Erreur lors du chargement d'un plugin."""
    pass

class PluginInitError(PluginError):
    """Erreur lors de l'initialisation d'un plugin."""
    pass

class PluginDependencyError(PluginError):
    """Dépendances manquantes ou incompatibles."""
    pass

class PluginVersionError(PluginError):
    """Version de l'API incompatible."""
    pass

class PluginPermissionError(PluginError):
    """Permission refusée pour une opération."""
    pass
```

6. Exemple de Plugin Complet

6.1 Plugin "Météo" (`plugins/meteo/__init__.py`)

```
python
```

```
from PyQt6.QtWidgets import QAction, QInputDialog
from PyQt6.QtGui import QIcon
from core.plugin_api import BlueNotebookPlugin
import requests
```

```
class MeteoPlugin(BlueNotebookPlugin):
```

```
    """Plugin pour insérer la météo actuelle."""
```

```
    def on_initialize(self):
```

```
        """Initialisation du plugin."""
```

```
        self.log_info("Initialisation du plugin Météo")
```

```
        # Charger la ville par défaut depuis les paramètres
```

```
        self.ville = self.get_settings('ville_defaut', 'Paris')
```

```
        self.api_key = self.get_settings('api_key', "")
```

```
    def register_actions(self):
```

```
        """Enregistre l'action dans le menu."""
```

```
        action = QAction("Insérer la météo", self.main_window)
```

```
        action.setIcon(QIcon(str(self.metadata['_path'] / 'assets' / 'icon.png')))
```

```
        action.setToolTip("Insère la météo actuelle pour une ville")
```

```
        action.triggered.connect(self.inserer_meteo)
```

```
        return [action]
```

```
    def register_preferences_widget(self):
```

```
        """Ajoute un onglet dans les préférences."""
```

```
        from .preferences_widget import MeteoPreferencesWidget
```

```
        widget = MeteoPreferencesWidget(self)
```

```
        return ("Météo", widget)
```

```
    def inserer_meteo(self):
```

```
        """Action principale : récupère et insère la météo."""
```

```
        # Demander la ville à l'utilisateur
```

```
        ville, ok = QInputDialog.getText(
```

```
            self.main_window,
```

```
            "Météo",
```

```
            "Entrez le nom de la ville :",
```

```

        text=self.ville
    )

    if not ok or not ville:
        return

    try:
        # Récupérer les données météo
        meteo_data = self._recuperer_meteo(ville)

        # Formater le texte
        texte = f"""
## Météo pour {ville}
- Température : {meteo_data['temperature']}°C
- Conditions : {meteo_data['conditions']}
- Humidité : {meteo_data['humidite']}%
"""

        # Insérer dans l'éditeur
        editor = self.get_editor()
        editor.insert_text_at_cursor(texte)

        self.show_notification("Météo", f"Météo insérée pour {ville}")

    except Exception as e:
        self.log_error(f"Erreur lors de la récupération de la météo : {e}", e)
        self.show_notification("Erreur", "Impossible de récupérer la météo")

    def _recuperer_meteo(self, ville: str) -> dict:
        """Récupère les données météo depuis une API."""
        if not self.api_key:
            raise ValueError("Clé API non configurée")

        # Appel API (exemple avec OpenWeatherMap)
        url = f"https://api.openweathermap.org/data/2.5/weather"
        params = {
            'q': ville,

```

```

        'appid': self.api_key,
        'units': 'metric',
        'lang': 'fr'
    }

    response = requests.get(url, params=params, timeout=5)
    response.raise_for_status()

    data = response.json()
    return {
        'temperature': data['main']['temp'],
        'conditions': data['weather'][0]['description'],
        'humidite': data['main']['humidity']
    }

def on_shutdown(self):
    """Nettoyage lors de la fermeture."""
    self.log_info("Arrêt du plugin Météo")

```

7. Intégration dans `main_window.py`

python

```
# Dans MainWindow.__init__()
from core.plugin_manager import PluginManager

# ... après la création des menus et de l'éditeur ...

# Initialiser le gestionnaire de plugins
self.plugin_manager = PluginManager(self, self.config_manager)
self.plugin_manager.initialize(self.editor, self.journal)

# Découvrir et charger les plugins
self.plugin_manager.discover_plugins()
self.plugin_manager.load_enabled_plugins()

# Dans MainWindow.closeEvent()
def closeEvent(self, event):
    """Événement de fermeture de la fenêtre."""
    self.plugin_manager.shutdown_all_plugins()
    # ... reste du code ...
    event.accept()
```

8. Gestion des Préférences

8.1 Ajout d'un Onglet Dynamique

Modifier `PreferencesDialog` pour supporter l'ajout dynamique d'onglets :

```
python
```

```

class PreferencesDialog(QDialog):
    """Dialogue des préférences avec support des plugins."""

    def __init__(self, parent=None):
        super().__init__(parent)
        self.plugin_tabs = {} # {plugin_id: (tab_index, widget)}
        # ... reste de l'initialisation ...

    def add_plugin_tab(self, plugin_id: str, tab_name: str, widget: QWidget):
        """Ajoute un onglet pour un plugin."""
        tab_index = self.tab_widget.addTab(widget, tab_name)
        self.plugin_tabs[plugin_id] = (tab_index, widget)

    def remove_plugin_tab(self, plugin_id: str):
        """Retire l'onglet d'un plugin."""
        if plugin_id in self.plugin_tabs:
            tab_index, widget = self.plugin_tabs[plugin_id]
            self.tab_widget.removeTab(tab_index)
            del self.plugin_tabs[plugin_id]

```

9. Sécurité et Bonnes Pratiques

9.1 Sandboxing (Optionnel pour v1)

Pour une première version, faire confiance aux plugins. Pour une version publique :

1. **Validation des permissions** : Vérifier que le plugin demande les permissions avant d'accorder l'accès
2. **Limitation des imports** : Interdire l'import de modules sensibles (`os.system`, `subprocess`, etc.)
3. **Timeout** : Limiter le temps d'exécution des méthodes des plugins
4. **Isolation des données** : Les plugins ne peuvent accéder qu'à leurs propres paramètres

9.2 Gestion des Erreurs

- Toujours encapsuler les appels aux plugins dans des `try/except`
 - Journaliser toutes les erreurs
 - Ne jamais laisser une erreur de plugin crasher l'application
 - Afficher un message clair à l'utilisateur en cas de problème
-

10. Tests et Validation

10.1 Tests